Project 2 Report - Predicting House Selling Prices

Group 6 - Krishnasai Chalasani & Shadman Hameed

ITSS 4V95.003 - AI & ML

11/13/2020

Table of Contents

1. Introduction/Problem Statement

Housing prices are of great interest for both buyers and sellers. Purchasing or selling a house is a big decision in anyone's life and as a result, it is always good to know how much a house is worth. Predicted housing prices can help buyers plan their finances and sellers know the trend of the market. We were given data on the specific details of residential homes in Ames, Iowa, such as house square feet, number of fireplaces, lot size, flatness of the property, type of dwelling, etc, and we were supposed to use these features to predict the selling price of these homes.

2. Importing Libraries

We first imported the necessary libraries and modules for this process. We primarily used pandas to clean the data and fill in missing values. We used scikit-learn and itertools to find the best combination of features. We used TuriCreate to create machine learning models and test their efficiency. We finally used matplotlib and seaborn to build our data visualizations.

3. Data Description

Overview of Dataset

We modified the testing dataset by adding the SalePrice column from the 'samplesubmission.csv' file to make evaluating our models easier. We first read in the training and testing datasets and saved both as pandas DataFrames. The training dataset consists of 1460 rows and 80 columns where the testing dataset consists of 1459 rows and 80 columns. The first column of both datasets is the index, so there are a total of 79 variables/features. Out of these features, there are 43 features with "object" data type and 36 quantitative or numerical features.

Exploratory Data Analysis

After looking at the overview of these 2 datasets, we then performed an exploratory data analysis using various methods like:

- .head() to print out the first 5 values

Id	MSSubClass	MSZoning	LotFrontage
1	60	RL	65.0
2	20	RL	80.0
3	60	RL	68.0
4	70	RL	60.0
5	60	RL	84.0

- .info() to provide a concise summary of the columns including their data types and the number of non-null values

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1460 entries, 1 to 1460
Data columns (total 80 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   MSSubClass     1460 non-null   int64
 1   MSZoning       1460 non-null   object
 2   LotFrontage    1201 non-null   float64
 3   LotArea        1460 non-null   int64
 4   Street         1460 non-null   object
 5   Alley          91 non-null     object
```

- .describe to show descriptive statistics like mean and standard deviation

	MSSubClass	LotFrontage	LotArea
count	1460.000000	1201.000000	1460.000000
mean	56.897260	70.049958	10516.828082
std	42.300571	24.284752	9981.264932
min	20.000000	21.000000	1300.000000
25%	20.000000	59.000000	7553.500000
50%	50.000000	69.000000	9478.500000
75%	70.000000	80.000000	11601.500000
max	190.000000	313.000000	215245.000000

This allowed us to better understand the underlying structure of both datasets.

4. Data Cleansing/Pre-Processing

<u>Dealing with Missing/Null Values</u>

Data cleansing is arguably the most important step in the data science process. One of the first things that we did was look at the data dictionary to become familiar with the column data types. What stood out was that there were a number of categorical columns that had an ‘NA’ category. The issue with this is that both the pandas and TuriCreate libraries will interpret these values as null and not include them in our later analysis. For example, if we look at the output of the .info method above, we see that there are only 91 non-null values in the Alley column. The

data dictionary has an ‘NA’ category that simply translates to No Alley. Similarly, there are other columns that suffer from this same issue.

To avoid this, our first action was to fill the false ‘NA’ records with a placeholder value in both the training and testing dataset for those categorical columns in the data dictionary that have a defined ‘NA’ value.

```
In [21]: # We will iterate through the list of columns_with_no_actual_null values and replace the NA values with a placeholder value
         for col in columns_with_no_actual_null_values:
             train_df[col].fillna('asdf', inplace = True)
```

At this point, we can be sure that any columns with detected null values actually have missing values.

For both the testing and training datasets, any columns that were category types with missing values were imputed with the most common values in that column. For some columns, the data was almost normally distributed as the mean and median were very close to each other so if the column had discrete values, we chose to impute the median, but if the column had continuous values, we chose to impute the mean. For other columns, there were some outliers and skewness as the mean and median were very different. As a result, we chose to replace the missing values with the median value as that is often regarded as the better choice.

Label Encoding

This last step is where we convert every column that has an object datatype into categories which are depicted as integer values. This method is referred to as label encoding. What we first do is create a list of columns that are currently of type object and convert it into a category type in the code below. The last cell then applies a function that changes the category columns into a numerical format.

```
In [51]: cat_columns = train_df.select_dtypes(['object']).columns

In [52]: cat_columns

Out[52]: Index(['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities',
                'LotConfig', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2',
                'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
                'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation',
                'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
                'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual',
                'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
                'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature',
                'SaleType', 'SaleCondition'],
               dtype='object')

In [53]: for col in cat_columns:
             train_df[col] = train_df[col].astype('category')
             test_df[col] = test_df[col].astype('category')

In [54]: train_df[cat_columns] = train_df[cat_columns].apply(lambda x: x.cat.codes)
         test_df[cat_columns] = test_df[cat_columns].apply(lambda x: x.cat.codes)
```

The reason why we do label encoding is that not only can we reap the benefits of having a compact data size and getting plotting support, but by converting categorical data into numerical data, our predictive models can better understand this data. Our machine learning algorithms can then decide in a better way on how those labels must be operated. It is regarded as an important preprocessing step for the structured dataset in supervised learning.

5. Finding Most Important Features

We used 2 methods of finding the most important features to improve model performance and avoid overfitting.

Univariate Selection

```
# Using SelectKBest class with chi-squared test (used to measure statistical significance) to retrieve the top 10 featu
bestfeatures = SelectKBest(score_func=chi2, k=10)
fit = bestfeatures.fit(X,y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(X.columns)

featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Specs','Score']
print(featureScores.nlargest(10,'Score'))  # Printing the 10 best features

first_set_of_important_features = list(featureScores.nlargest(10,'Score')['Specs']) # Retrieving the 10 best features
```

The first method is regarded as univariate selection. We used the SelectKBest class from the scikit-learn library, which conducts statistical tests like chi-squared, to find features that have the strongest relationship with SalePrice. We then selected the 10 features that have the highest importance scores and saved it in a list variable.

Specs	Score
LotArea	1.011497e+07
MiscVal	6.253332e+06
2ndFlrSF	4.648841e+05
BsmtFinSF1	3.999851e+05
PoolArea	3.835642e+05
BsmtFinSF2	3.688827e+05
MasVnrArea	2.880241e+05
BsmtUnfSF	2.747512e+05
LowQualFinSF	2.448810e+05
GrLivArea	1.968501e+05

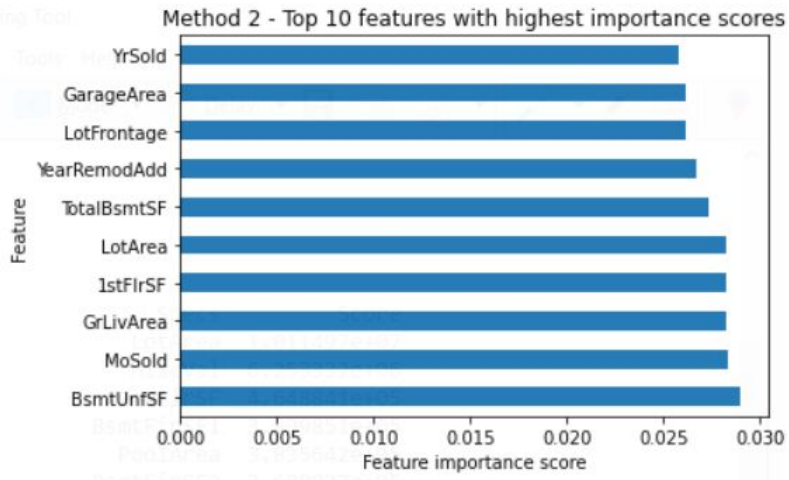
Using the Extra Trees Classifier

```
model = ExtraTreesClassifier()
model.fit(X,y)

feat_importances = pd.Series(model.feature_importances_, index=X.columns) # Using inbuilt class feature_importances of tre
feat_importances.nlargest(10).plot(kind='barh') # Plotting bar chart of feature importance
plt.title('Method 2 - Top 10 features with highest importance scores')
plt.xlabel('Feature importance score')
plt.ylabel('Feature')
plt.show()

second_set_of_important_features = list(feat_importances.nlargest(10).index) # Retrieving features that have the 10 highes
```

The second method uses the feature importance class, which is inbuilt and comes with Tree Based Classifiers such as the Extra Tree Classifier. We then plotted and retrieved the top 10 features and saved it in another list variable.



Finally, we made a cumulative set of unique, important features from these 2 methods and used this for our feature list for some of the models.

```
# Getting a cumulative list of unique, important features from the 2 ways

cumulative_list_important_features = first_set_of_important_features + second_set_of_important_features
cumulative_set_important_features = set(cumulative_list_important_features)
print("Cumulative set of unique, important features from the two ways:", cumulative_set_important_features)
```

```
Cumulative set of unique, important features from the two ways: {'MasVnrArea', 'PoolArea', 'BsmtFinSF2', 'MiscVal', 'GrLivAr
ea', 'MoSold', 'GarageArea', 'LotArea', 'LotFrontage', '1stFlrSF', 'BsmtUnfSF', '2ndFlrSF', 'TotalBsmtSF', 'BsmtFinSF1', 'Ye
arRemodAdd', 'LowQualFinSF', 'YrSold'}
```

6. Regression Techniques

Generic Regression Model

We first created a baseline regression model where we don't specify any features. We did this so that we have a comparison between this and our later regression models. This model yielded a root mean squared error (RMSE) value of nearly 65,000.

```
In [60]: regression_model = tc.regression.create(train_SF_regression, target='SalePrice')
         regression_model_errors = regression_model.evaluate(test_SF_regression)
         print(regression_model_errors)
```

```
| 2         | 0.058183     | 497678.750000      | 254459.046875        | 101919.429688
|
| 3         | 0.081477     | 408979.375000      | 195411.859375        | 74188.937500
|
| 4         | 0.103347     | 328381.562500      | 161758.265625        | 54703.546875
|
| 5         | 0.128506     | 267127.218750      | 138194.375000        | 41051.800781
|
| 10        | 0.248482     | 95691.625000       | 108779.343750        | 14766.959961
|
+-----------+--------------+--------------------+----------------------+-----------------------
--------------------+

{'max_error': 313203.296875, 'rmse': 64659.06734252475}
```

Linear Regression Model

We then decided to implement linear regression as our hypothesized primary model. Since we were trying to predict SalePrice, which was a continuous value and a known output, this use-case was ideal for linear regression.

- Find ideal combination of features

Our goal was to now find which combination of features would make a linear regression model generate the lowest RMSE. Using an outer for-loop which iterated three times, combinations of features were generated, iterated, and tested for the lowest RMSE. Once the inner loop was completed, we appended that combination of features to our final list of important features.

Finally, since we wanted to find the single best combination of features and their respective errors, we ran a final for-loop that finds the lowest RMSE value which we then used to recreate our ideal linear regression model.

```
ideal_linear_regression_model_features_and_errors = []

for i in range(1, 4):
    list_combination_indexes = list(it.combinations(list(cumulative_set_important_features), i))
    ideal_linear_regression_model_features = list(list_combination_indexes[0])
    linear_regression_model = tc.linear_regression.create(train_SF_regression, target = 'SalePrice', features = ideal_linear_
    ideal_linear_regression_model_errors = linear_regression_model.evaluate(test_SF_regression)

    print('BEGINNING')
    print('Number of features used:', len(ideal_linear_regression_model_features))
    print('Features used:', ideal_linear_regression_model_features)
    print('Errors:', ideal_linear_regression_model_errors)

    for j in list_combination_indexes:
        linear_regression_model = tc.linear_regression.create(train_SF_regression, target = 'SalePrice', features = list(j))
        linear_regression_model_errors = linear_regression_model.evaluate(test_SF_regression)

        if (linear_regression_model_errors['rmse'] < ideal_linear_regression_model_errors['rmse']):
            ideal_linear_regression_model_features = list(j)
            ideal_linear_regression_model_errors = linear_regression_model_errors

    print('ENDING')
    print('Number of features used:', len(ideal_linear_regression_model_features))
    print('Features used:', ideal_linear_regression_model_features)
    print('Errors:', ideal_linear_regression_model_errors)

    ideal_linear_regression_model_features_and_errors.append([ideal_linear_regression_model_features, ideal_linear_regression_

print(ideal_linear_regression_model_features_and_errors)

final_ideal_linear_regression_model_features_and_errors = ideal_linear_regression_model_features_and_errors[0]

for i in ideal_linear_regression_model_features_and_errors:
    if i[1] < final_ideal_linear_regression_model_features_and_errors[1]:
        final_ideal_linear_regression_model_features_and_errors = i

print(final_ideal_linear_regression_model_features_and_errors)
```

- Coefficients

  The coefficients for this linear regression model are listed below:

```
In [72]: ideal_linear_model.coefficients
```

Out[72]:

name	index	value	stderr
(intercept)	None	286431.1321370237	3175749.649006359
LotArea	None	2.0810014562765553	0.20462917199251393
MoSold	None	1207.6020084944478	782.6753464496375
YrSold	None	-67.03927157406022	1581.3158112287629

[4 rows x 4 columns]

For one unit of change in the feature variables LotArea, MoSold, and YrSold, there is a corresponding change in the predictor variable SalePrice which is represented by the magnitude of the coefficients.

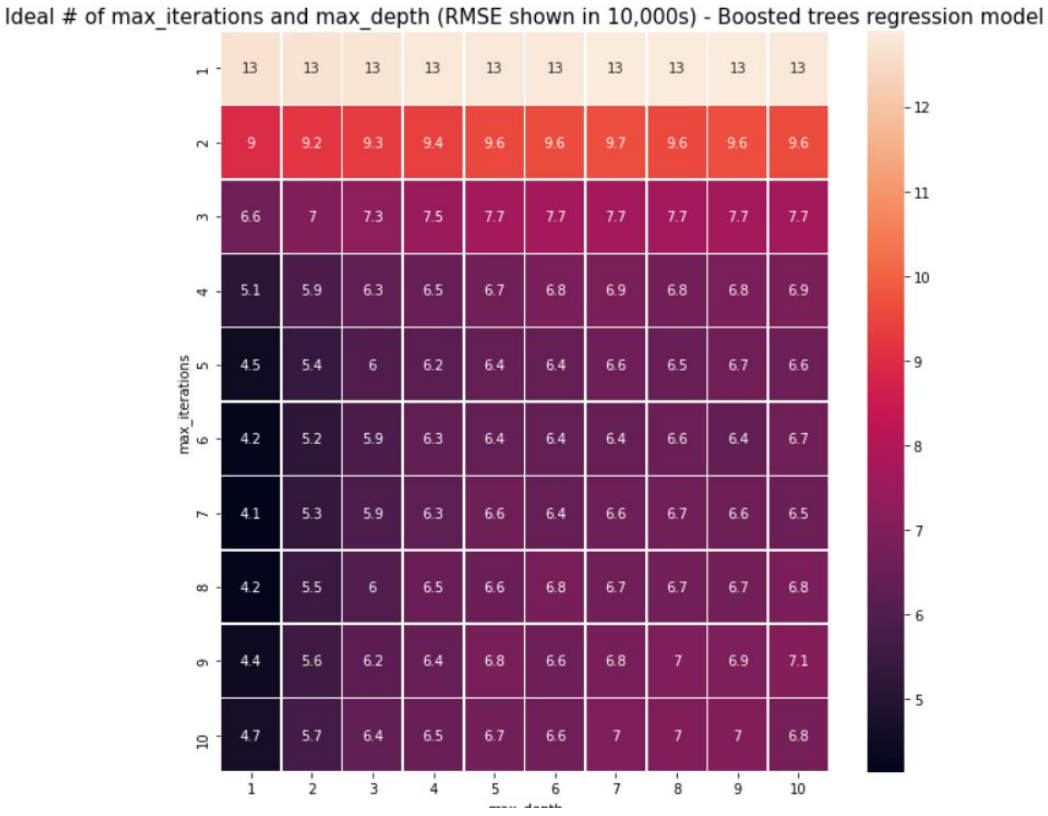
Boosted Tree Regression Model

- Model description

We chose to create a boosted tree regression model. A boosted tree regression model is an ensemble method, which combines predictions from multiple ML algorithms (decision trees) together to form its predictions. It repeatedly fits many decision trees to improve the accuracy of the model. The model selects the data to build the trees through the boosting method where the input data have weights. The way the weights are selected is that if data was not used well in one tree, that data has a higher chance of being selected in a new tree. The model continuously tries to improve its accuracy by using the fit and the prediction errors of previous trees in the subsequent trees.

- Tuning the model

We tried to find ideal values for 2 of the parameters for the model, max_depth and max_iterations, which will cause the model to generate the least amount of RMSE. We created a loop that iterates through every combination of max_depth and max_iterations up till a value of 10 for each parameter due to computation expense and plotted a heatmap showing each combination as well as the model's corresponding RMSE.



As seen in the heatmap, the ideal combination of max_depth and max_iterations that will output the least amount of RMSE is if max_depth = 1 and max_iterations = 7. We then saved these two values as the ideal parameter combination for the boosted tree regression model.

7. Classification Techniques

Preparation for Classification

Unlike regression, where our model would predict the exact House Price, the classification models required us to change our target from a continuous SalePrice value to discrete, binned values.

To do this, we created bins by first finding the maximum and minimum values between the testing and training datasets and determining the largest range. The reason we did this is that we wanted to be able to train the models so that they could predict bin values in both the testing and training datasets.

The code below shows the creation of the bins (size 10,000), applying those bins into a new column in both the testing and training sets, and finally dropping the SalePrice field from the newly created SFrames so that our classification models don't use SalePrice as a feature in determining the associated bin values.

```
In [65]: max_value = max(train_df['SalePrice'])
         min_value = min(train_df['SalePrice'])

         if max(test_df['SalePrice']) > max_value:
             max_value = max(test_df['SalePrice'])

         if min(test_df['SalePrice']) < min_value:
             min_value = min(test_df['SalePrice'])

         bins = []

         for i in range(min_value - 10000, max_value + 10001, 10000):
             bins.append(i)

         labels = list(range(1, len(bins)))

         train_df['SalePrice_bin_range'] = pd.cut(train_df['SalePrice'], bins = bins)
         train_df['SalePrice_label_for_bin_range'] = pd.cut(train_df['SalePrice'], bins = bins, labels = labels)

         label_for_bin_range_SalePrice = dict(zip(list(train_df['SalePrice_label_for_bin_range']), list(train_df['SalePrice_bin_range']

         test_df['SalePrice_bin_range'] = pd.cut(test_df['SalePrice'], bins = bins)
         test_df['SalePrice_label_for_bin_range'] = pd.cut(test_df['SalePrice'], bins = bins, labels = labels)

         train_df.drop('SalePrice_bin_range', inplace = True, axis = 1)
         test_df.drop('SalePrice_bin_range', inplace = True, axis = 1)
```

```
In [66]: train_df['SalePrice_label_for_bin_range'] = train_df['SalePrice_label_for_bin_range'].astype(int)
         test_df['SalePrice_label_for_bin_range'] = test_df['SalePrice_label_for_bin_range'].astype(int)

         train_df_classifier = train_df.drop('SalePrice', axis = 1)
         test_df_classifier = test_df.drop('SalePrice', axis = 1)

         train_SF_classifier = tc.SFrame(train_df_classifier)
         test_SF_classifier = tc.SFrame(test_df_classifier)
```

Generic Classification Model

Like for regression, we first created a generic classification model to see what model TuriCreate will automatically choose and its respective accuracy. Before evaluating the model with the testing dataset, based on the accuracy, we saw that the boosted tree and random forest

classifier models were almost consistently the top 2 performers each time the generic model was implemented.

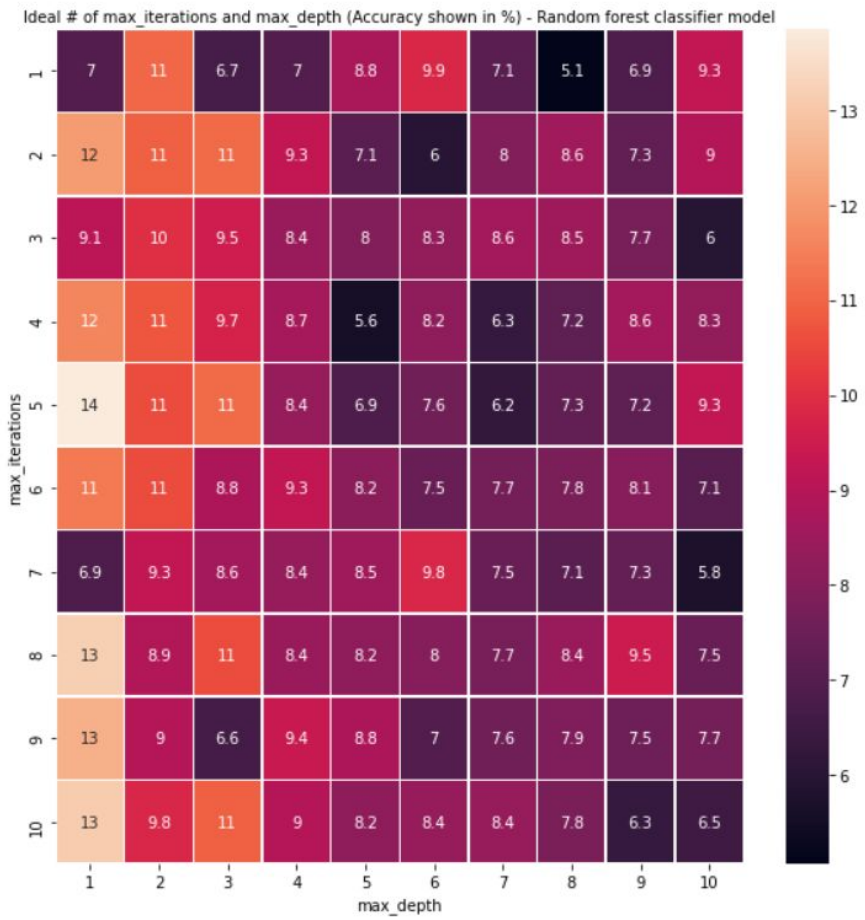
Random Forest Classification Model

- Model description

We decided to proceed with creating a random forest classifier model. A random forest classifier model is also an ensemble method. It is similar to a boosted tree model but the difference between the two models lies in the ways the models select the data. While the boosted tree model selects the input data based on applying weights to input data, the random forest model selects the input data through the bootstrap aggregation or bagging technique where each piece of data is equally likely to be selected for the decision trees to train on and form predictions. These predictions are then averaged to form a random forest prediction.

- Tuning the model

Similar to the model tuning procedure for the boosted tree regression model, we tried to find ideal values for max_depth and max_iterations that will cause the model to generate the most amount of accuracy after evaluating the model with the testing dataset. We tried to loop through every combination of max_depth and max_iterations up till a value of 10 for each parameter due to computation expense and plotted a heatmap showing each combination as well as the model's corresponding accuracy after evaluation.

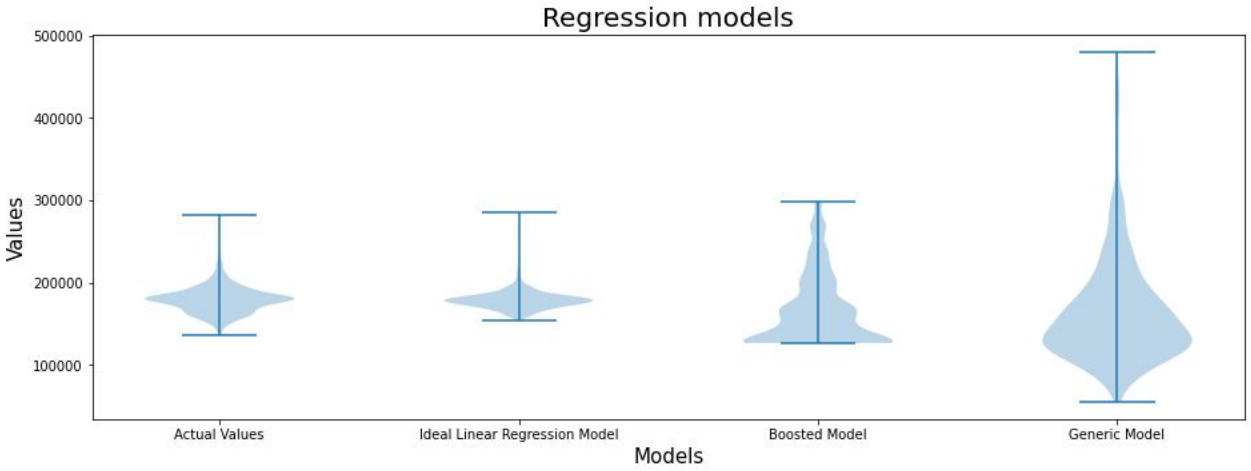


As seen in the heatmap, the ideal combination of max_depth and max_iterations that will output the most amount of accuracy is if max_depth = 1 and max_iterations = 3. We then saved these two values as the ideal parameter combination for the random forest classifier tree model.

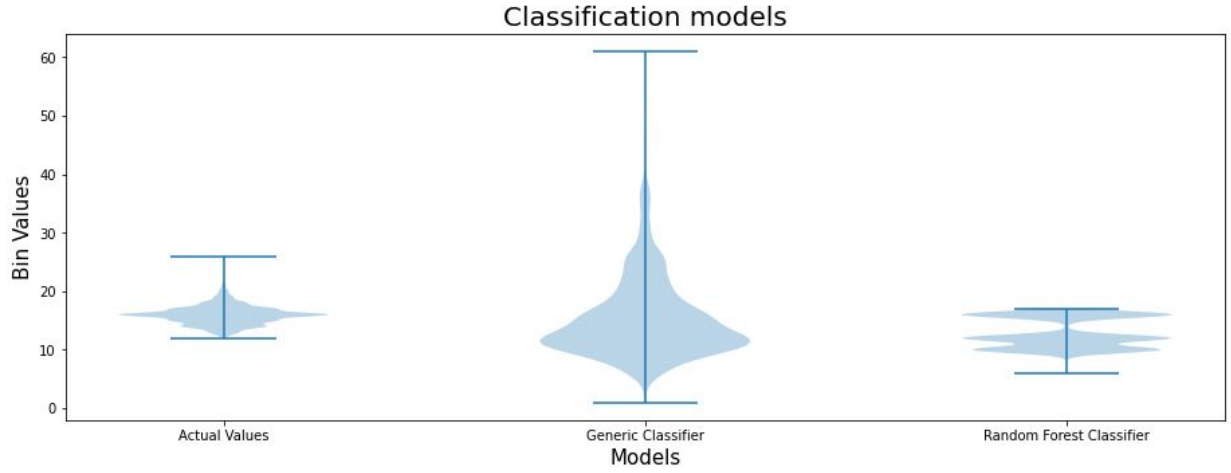
8. Results and Analysis

Violin Plots

The first series of graphs that we have are violin plots. We decided to compare the distributions of our models' predicted SalePrice to that of the actual data (testing data-SalePrice). Although this plot can't compare individual predictions to their actual values, it's useful to see a high level comparison of the SalePrice distributions. It's important to note that among our regression models tested, the distribution of our ideal linear regression model's predicted SalePrice most closely matches with that of the actual values in the testing dataset. Our optimized boosted tree model was not nearly as accurate in its predictions, but was able to infer a similar range of predicted values. The generic linear regression model created below predicted prices that were far beyond the range of the actual values.

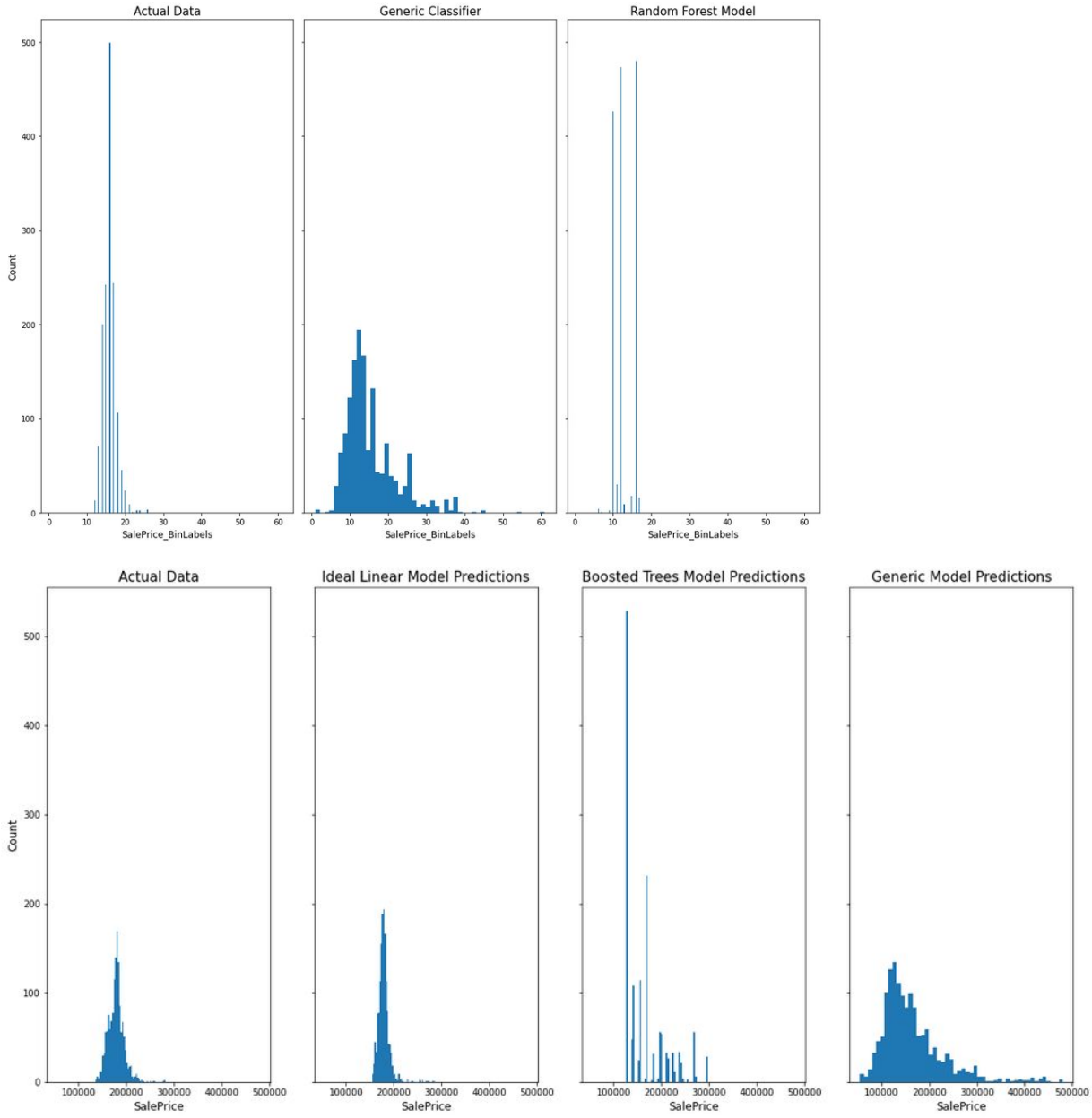


Below, we can see the results of our classification models as well. Both our generic model and optimized random forest classifier model were not able to accurately predict the correct SalePrice bin values, but our random forest classifier was more accurate. However, it is important to note that since the bins already have an inherent error of 10,000, the classification model is likely not the best way to approach this problem.



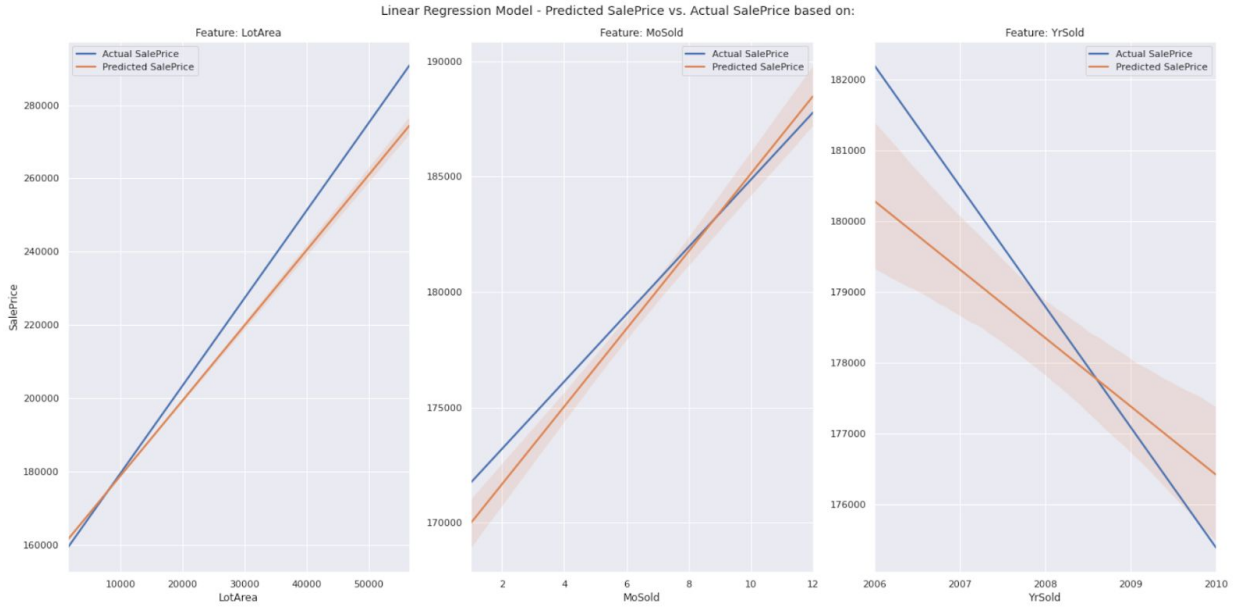
Histogram Plots

Another visualization of our models' predictions that we can use are histograms. These show in greater detail the distribution of both our regression and classification models. The reason that the bars of the classification histograms were not contiguous is because the histogram was attempting to bin values that are already binned. As seen in the violin plots, we noticed very similar results in the histograms as the ideal linear regression and tuned random forest regression models seemed to perform the best for regression and classification respectively.

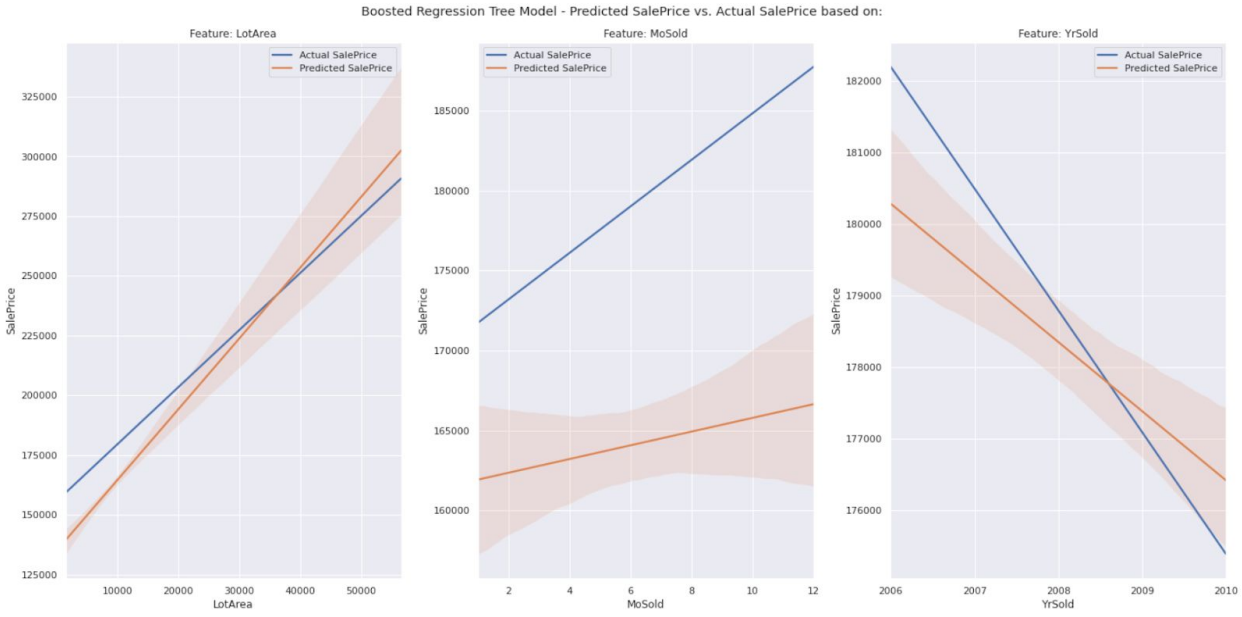


Line Plots - Regression

We built a cumulative line plot comparing predicted SalePrice with actual SalePrice based on the three ideal features, LotArea, MoSold, and YrSold, for the linear regression model.



As seen in the plots, we can argue that the lines of best fit for the predicted SalePrice closely match the lines of best fit for the actual SalePrice based on these three features. We also built a line plot comparing predicted SalePrice with actual SalePrice based on the three features for the boosted tree regression model.

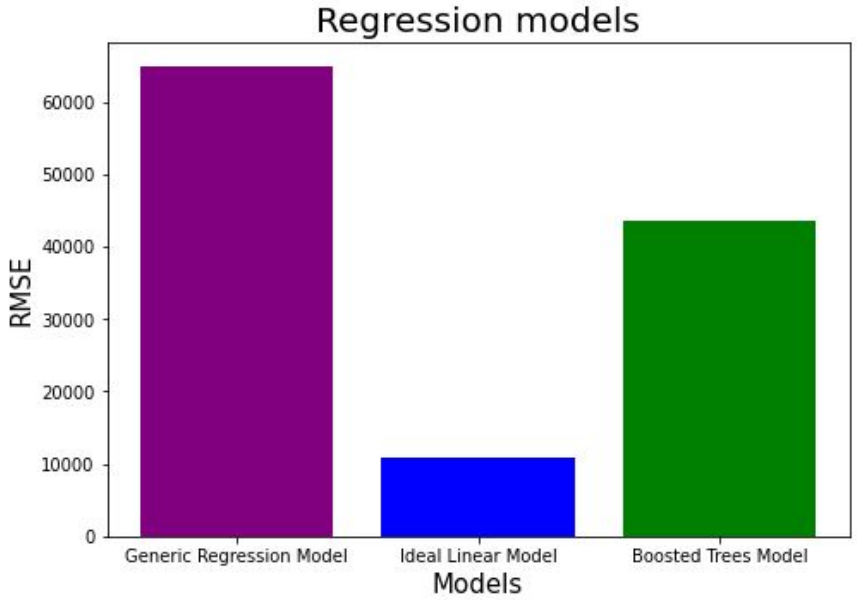


As seen in the plots, except for the MoSold feature, we can argue that the lines of best fit for the predicted SalePrice closely match the lines of best fit for the actual SalePrice based on the other two factors.

Furthermore, we can conclude that the linear regression model using LotArea, MoSold, and YrSold is the ideal regression model to use for this situation.

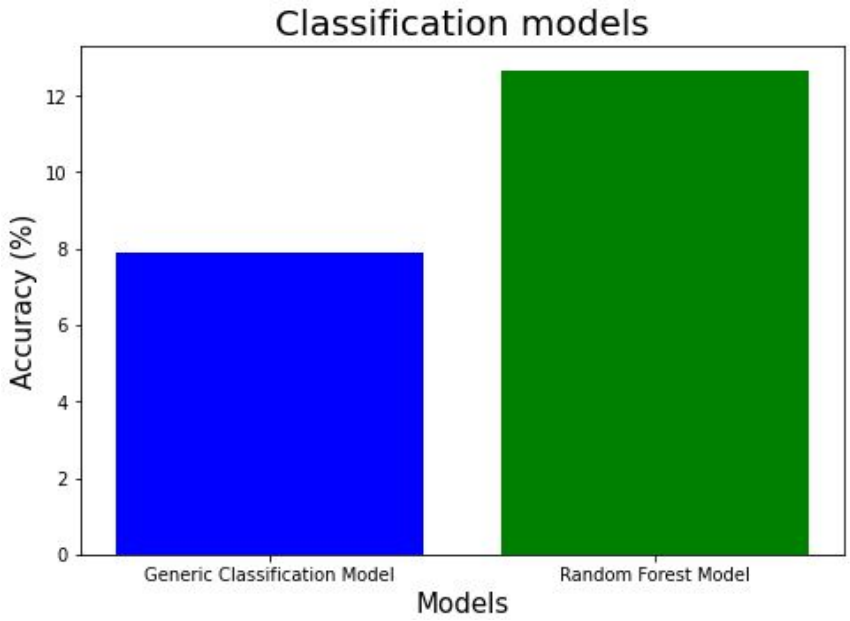
9. Summary/Conclusion

We first measured the performance of all 3 regression models (generic, linear regression, and boosted tree regression) and displayed the RMSE value of each model after evaluation.



Looking at the bar chart, we can see that the two models that we had built are far better than the generic regression model. Concretely, the linear regression model with the ideal feature combination performed the best with a RMSE value of roughly only 11,000.

We then measured the performance of both classification models and plotted the accuracy of each model after evaluation.



Looking at the bar chart, we can see that the random forest classifier with the ideal max number of iterations and depth performed better than the generic classifier model.

In conclusion, seeing that the very nature of this challenge is oriented towards the supervised machine learning method, linear regression, as we were trying to predict a known output of a continuous variable, SalePrice, we can argue that the linear regression model using LotArea, MoSold, and YrSold performed the best and most accurately predicted the selling price of these houses out of all the models built.

10. Future Steps/Areas of Improvement

All of our tuned models performed better than our base models, but there are various areas of where our models can improve:

Cleaning the Data

For the models that we had built, we analyzed the distribution of columns with missing values and imputed the mean, median, or mode of each column appropriately. As a future step, we could use machine learning to impute missing values through methods like k-nearest neighbours or deep learning, which could prove to be more accurate ways of cleaning the data.

Finding Best Features

- We used 2 different methods where each method found the 10 best features and combined both methods into a unique list of best features.

- We could explore different ways of finding strongly correlated features with SalePrice such as Lasso and RF, which are embedded methods, or recursive feature elimination.
- We could increase the number of best features that each method outputs.

Building Machine Learning Models

- Regression
  - Linear regression
    - When the model is created, TuriCreate randomly selects 5% of the training dataset and assigns it to the validation dataset if the validation dataset is not specified. We would explore the impact of changing the size of the validation dataset on the RMSE value of the linear regression model.
    - Due to the amount of computation expense associated with finding the ideal combination of features, we put a limit on the number of features as 3. However, if we are working with a more powerful computer that can deal with more amounts of payload, we would explore the impact of changing the limit to more than 3 features on the RMSE value of the linear regression model.
    - Boosted tree regression
      - We would try using different parameters such as step_size, which is the inverse of max_iterations, and min_child_weight, which is the number of observations at each node of the tree, in tuning the model.
    - We would try using different regression models such as decision tree regression and random forest regression to compare the RMSE and max_error values of each model and do a complete analysis to see which model performs the best.

- Classification
  - Random forest classification
    - We would try using different parameters such as num_trees, which is the number of trees used in the model, and step_size to see the impact on the model's accuracy.
  - We would try using different classification models such as logistic regression and nearest neighbor classification to compare the error and accuracy of each model after evaluation.
- We would examine different clustering methods like k-means to be used on the training dataset to group similar houses, predict which clusters the homes in the testing dataset would fall under, and look at the clusters' price range. We would also look at the

accuracy and error of using these methods and compare it with those of the supervised machine learning models.

Works Cited

Learn. (n.d.). Retrieved from https://turi.com/learn/

Pandas. (n.d.). Retrieved from https://pandas.pydata.org/

Statistical data visualization. (n.d.). Retrieved from https://seaborn.pydata.org/

Towards Data Science. (n.d.). Retrieved from https://towardsdatascience.com/

Visualization with Python. (n.d.). Retrieved from https://matplotlib.org/

Where Developers Learn, Share, & Build Careers. (n.d.). Retrieved from https://stackoverflow.com/

Where good ideas find you. (n.d.). Retrieved from http://medium.com/